Sémantika programovacích jazyků

Ing. Stanislav Elbl
VUT Brno, Fakulta informačních technologií

PDF created with FinePrint pdfFactory trial version http://www.fineprint.com

Obsah

1. ÚVOD 3

2. SÉMANTIKA 3

2.1. Atributy, vazba, sémantické funkce 3
2.2. Deklarace, bloky, rozsah platnosti 4
2.3. Tabulka symbolů 5
2.4. Prostředí 6
2.5. Proměnné a konstanty 7
2.6. Alias, neplatná reference 7
2.7. Vyhodnocování výrazů 8

3. FORMÁLNÍ SPECIFIKACE SÉMANTIKY 9

3.1. Operační sémantika 9
3.2. Denotační sémantika 11
3.3. Axiomatická sémantika 12

4. LITERATURA 13

PDF created with FinePrint pdfFactory trial version http://www.fineprint.com

1. Úvod

Algoritmus zapsaný v programovacím jazyku musí mít přesně dané chování, které lze odvodit z jeho zápisu. To vyžaduje přesnou definici jazyka. První částí jeho definice je specifikace syntaxe. Tu je možné popsat pomocí gramatik - většinou se používají bezkontextové gramatiky. S tím souvisí také lexikální struktura jazyka, která je také specifikována gramatikami. Druhou, nedílnou součástí definice jazyka je jeho sémantika (význam, chování). Tento článek pojednává o základních pojmech souvisejících se sémantikou a o jejím formálním popise.

2. Sémantika

Pro úplnou definici jazyka nestačí popsat jeho syntaxi. Ta definuje pouze jeho strukturu. Aby byla zajištěna přenositelnost programů a jejich nezávislost na implementaci překladače, je nutné specifikovat význam jednotlivých konstrukcí. Nejjednodušší metodou je slovní popis v přirozeném jazyku. Příkladem může být následující popis syntaxe a sémantiky podmíněného příkazu v jazyku Pascal:

```
if VÝRAZ then PŘÍKAZ1 [ else PŘÍKAZ2 ];
```

> Při vykonávání podmíněného příkazu se nejdříve vyhodnotí VÝRAZ. Pokud je pravdivý, provede se PŘÍKAZ1. Pokud je nepravdivý a podmíněný příkaz obsahuje část uvozenou klíčovým slovem **else**, provede se PŘÍKAZ2.

Takový popis sémantiky se používá v referenčních manuálech a v učebnicích konkrétních jazyků. Někdy je však nutný matematicky přesný popis významu - například pro dokazování algoritmů je to nezbytné. Proto existují také formální specifikace jako je operační, axiomatická nebo denotační sémantika.

V této kapitole budou vysvětleny důležité základní rysy sémantiky programovacích jazyků a principy s ní spojené, v následující kapitole pak její formální modely.

2.1. Atributy, vazba, sémantické funkce

Základním mechanismem abstrakce ve většině programovacích jazyků je používání jmen (**identifikátorů**) označujících jednotlivé entity. Popis sémantiky spočívá v přiřazení významu jednotlivým identifikátorům.

Dalšími pojmy jsou **umístění** a **hodnota**. Hodnoty představují například celá čísla nebo reálná čísla. Umístění jsou místa, kde mohou být hodnoty uloženy. Jsou to adresy paměťových buněk v počítači nebo obecněji například čísla.

Význam identifikátoru je určen pomocí **atributů**. Atributem může být již zmíněné umístění nebo hodnota, ale také určení, že identifikátor představuje konstantu, proměnnou nebo funkci (včetně jejich parametrů), a dále datový typ. Atributy mohou být přiřazeny deklarací nebo přiřazením.

Přiřazení atributu identifikátoru se nazývá vazba. Atribut může být definován v různých časech - při překladu nebo linkování programu (**statická vazba**) a během běhu programu (**dynamická vazba**). Podle toho se rozlišují statické a dynamické atributy - čas vazby atributu je

PDF created with FinePrint pdfFactory trial version http://www.fineprint.com

však závislý na použitém programovacím jazyku a také na konkrétní implementaci překladače. Například u interpretů jsou vazby prováděny dynamicky. Kromě toho lze rozlišit ještě podkategorie - vazba při definici jazyka, v době implementace, v době překladu, při linkování, při zavedení programu a v době běhu.

Například u deklarace

```
var i : integer;
```

překladač přiřadí identifikátoru `i` datový typ a zaznamená si, že se jedná o proměnnou. Dále mu v době linkování nebo zavedení programu přiřadí umístění.

Překladač musí evidovat všechny atributy a vazby. K tomu slouží **tabulka symbolů**. Kompilátor uchovává v tabulce symbolů pouze statické atributy, zatímco interpret v ní uchovává všechny atributy. Matematicky představuje tabulka symbolů funkci z množiny identifikátorů do množiny atributů. Další důležitá funkce přiřazuje identifikátoru umístění v paměti - označuje se jako **prostředí** (environment). **Paměť** potom představuje funkci přiřazující danému umístění konkrétní hodnotu.

2.2. Deklarace, bloky, rozsah platnosti

K vytvoření vazby slouží **deklarace**. Některé jazyky používají **explicitní** deklarace (například Pascal), jiné **implicitní** - pouhým prvním použitím daného identifikátoru. U implicitních bývá nutné dodržet doplňující pravidla pro vytváření jmen.

Deklarace jsou spojeny s **blokem**. Například v Pascalu mohou být **globální deklarace** definované v rámci programu a **lokální deklarace** spojené s konkrétní funkcí, ve které se vyskytují. Bloky určují rozsah platnosti identifikátorů - místa v programu, kde je identifikátor platný a má určité atributy. Jeden identifikátor může mít na různých místech přiřazeny jiné atributy, což znamená, že může mít jiný význam. Jazyky, v nichž mohou být bloky zanořovány se nazývají blokově strukturované jazyky.

Následující příklad obsahuje deklarace s různým rozsahem platnosti:

```
program RozsahPlatnosti;

var x : integer;  _______________
                                 |
procedure p;   ______________    |
var y : integer;  ___        |   |
begin                |       |   |
   …                 |       |   |
end;                 ↓       |   |
                             |   |
procedure q;  ___________    |   |
var x : boolean;  ___    |   |   |
begin                |   |   |   |
   …                 |   |   |   |
end;                 ↓   |   |   |
                         |   |   |
begin                    |   |   |
   …                     |   |   |
end;                     ↓   ↓   ↓
```

PDF created with FinePrint pdfFactory trial version http://www.fineprint.com

Platí zde, že identifikátor je platný v bloku, v němž je deklarován od místa deklarace dále (v jiných jazycích to může být jinak). Deklarace ve vnořených blocích mají vyšší prioritu než deklarace v nadřazených blocích. V příkladu to ilustruje identifikátor `x` - ve funkci `p` je to lokální proměnná typu boolean, přestože dříve (v nadřazeném bloku) již byla definována jiná proměnná se stejným názvem. Někdy se proto rozlišuje platnost a **viditelnost** deklarace. Viditelnost zahrnuje pouze ta místa v programu, kde je daný identifikátor přístupný - globální proměnná `x` v příkladu není přístupná ve funkci `q`.

2.3. Tabulka symbolů

Tabulka symbolů může být implementována pomocí různých datových struktur a jejich kombinací (tabulky, vyhledávací stromy, seznamy). Bloková struktura programů však vyžaduje využití zásobníku. Nejdříve totiž musí být zpracovány deklarace vnořeného bloku. Na jeho začátku jsou přidány do tabulky symbolů a na konci jsou z ní vyjmuty. Obecně se může na tabulku symbolů pohlížet jako na soubor jmen. Ke každému jménu je v ní asociován zásobník deklarací na jehož vrcholu je deklarace aktivní v daném místě programu.

Existují dva rozdílné rozsahy platnosti - dynamický a statický. Na tom, který z nich se v daném jazyku používá závisí implementace tabulky symbolů. V případě statického je platnost identifikátorů omezena blokem, v němž se vyskytuje jeho deklarace. Na rozdíl od toho je u dynamického rozsahu (tabulka symbolů musí být v tomto případě vytvářena při běhu programu) platnost identifikátorů dána cestou běhu programu - tak jak se postupně vyskytly deklarace v jeho kódu. Následující příklad ilustruje rozdíl mezi dynamickým a statickým rozsahem platnosti:

```
var x : integer;

procedure p;
begin
   writeln(x);
end;

procedure q;
var x : integer;
begin
   x:=2;
   p;
end;

begin
   x:=1;
   p;
   q;
end;
```

V případě statického rozsahu platnosti program vytiskne dvakrát po sobě hodnotu globální proměnné x - tedy 1. Pokud by byl použit dynamický rozsah (to v jazyku Pascal samozřejmě není), program by nejdříve vytiskl hodnotu globální proměnné x, ale potom by vytisknul hodnotu lokální proměnné x z procedury q (protože v druhém případě je procedura p volána z q).

Problémem u dynamického rozsahu platnosti je například to, že chování procedury může být závislé na tom, z kterého místa programu je volána. V případě použití identifikátoru není totiž možné určit deklaraci použitého identifikátoru pouhým přečtením dané části kódu. Proto se většinou používá statický rozsah platnosti.

PDF created with FinePrint pdfFactory trial version http://www.fineprint.com

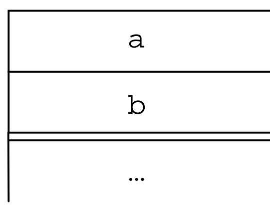
2.4. Prostředí

Prostředí spravuje vazby identifikátorů na jejich umístění. Opět může být statické (všechny identifikátory jsou umístěny před spuštěním programu - obvykle při jeho zavedení) nebo dynamické, případně kombinace obou. Například globální proměnné mohou být alokovány staticky, zatímco lokální proměnné se alokují dynamicky.

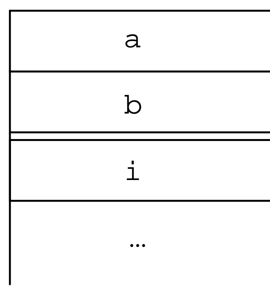
Podobně jako tabulka symbolů, používá i prostředí blokově strukturovaných jazyků zásobníkovou strukturu. V následujícím příkladě je vidět, jak postupně dochází k alokaci proměnných:

```
procedure p1;
var i : integer;
begin
   …
end;

procedure p2;
var a,b : integer;
begin
   …
   p1;
   …
end;

begin
   …
   p2;
   …
end;
```

a) Prostředí po vstupu do p2



b) Prostředí po vstupu do p1



Lokální proměnné procedury p2

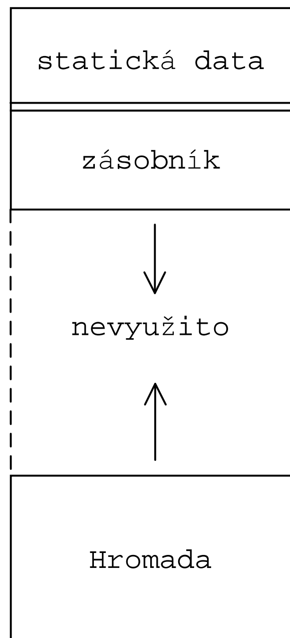
Lokální proměnné p1

Nevyužitá paměť

Při vstupu do bloku se v prostředí alokuje místo pro lokální proměnné a při výstupu z tohoto bloku se místo opět uvolní. Po ukončení procedury p1 bude tedy prostředí vypadat jako na obrázku a). Volání procedury nebo funkce se říká **aktivace** procedury a region paměti alokované při aktivaci se nazývá **aktivační záznam**.

Kromě globálních a lokálních proměnných existují také dynamické proměnné. Paměť je pro ně alokována v době běhu programu. Prostředí musí obsahovat oblast, v níž je tato paměť alokována (k alokaci a uvolnění této paměti slouží např. v Pascalu procedury new a dispose). Tato oblast se nazývá **halda** (nebo také hromada). Zásobník pro alokaci lokálních proměnných a halda se obvykle rozšiřují proti sobě na opačných koncích paměti jak naznačuje následující obrázek.

PDF created with FinePrint pdfFactory trial version http://www.fineprint.com



2.5. Proměnné a konstanty

Proměnná je objekt, jehož hodnota se může za běhu programu měnit. K tomu slouží příkaz **přiřazení** x := *e*, kde x je proměnná a *e* je výraz. Výraz je nejdříve vyhodnocen a výsledná hodnota je zkopírována na místo proměnné x. Je nutné rozlišovat umístění proměnné (tzv. l-hodnota) a její hodnotu. V některých jazycích může mít přiřazení jiný význam - nejsou kopírovány hodnoty, ale umístění (tedy ukazatel na danou hodnotu). Například v případě příkazu x := y dojde ke zkopírování umístění proměnné y do proměnné x - jedná se o přiřazení sdílením.

Konstanta má během provádění programu neměnnou hodnotu. Na rozdíl od proměnné nemusí mít přiřazeno umístění v paměti. Hodnota statické konstanty je vypočtena před spuštěním programu, zatímco hodnota dynamické konstanty je určena až při jeho běhu.

2.6. Alias, neplatná reference

Pokud různé identifikátory představují jeden objekt (například proměnnou), nazýváme je **aliasy**. Následující příklad ilustruje takovou situaci:

```
type pinteger = ^integer;
var x, y : pinteger;

begin
   new (x);
   x^ := 1;
   y := x;         { x a y jsou nyní aliasy }
   y^ := 2;
   writeln(x^);    { zde je vypsána hodnota 2 }
end;
```

Alias může být nebezpečný protože způsobuje **vedlejší účinky** (side effect) - může změnit hodnotu proměnné, která se nevyskytuje v příkazu přiřazení. Význam takového přiřazení není zřejmý pouze z jeho zápisu, ale musí se přečíst větší část programu.

PDF created with FinePrint pdfFactory trial version http://www.fineprint.com

Dalším problémem při používání ukazatelů jsou **neplatné reference** - místa, která byla z prostředí dealokována, ale v programu jsou stále přístupná. Pokud předchozí příklad změníme například takto:

```
new(x);
y:=x;
dispose(y);
writeln(x^); { zde se přistupuje k dealokované paměti }
```

dojde na posledním řádku k přístupu k dealokované paměti (a pravděpodobně k havárii programu). K podobnému výsledku může dojít i s lokálními proměnnými.

Třetím problémem jsou bloky paměti, na které nevede žádná reference (označované jako **garbage**). To nastane v případě, že se neuvolní alokovaná paměť. Činnost takového programu je sice správná, ale může se stát, že zhavaruje z důvodu nedostatku paměti. Z toho důvodu se v některých jazycích používá automatická dealokace paměti - garbage collector.

2.7. Vyhodnocování výrazů

V programovacích jazycích se rozlišuje příkaz a výraz. Výsledkem výrazu je hodnota a jeho vyhodnocení nemá žádné vedlejší účinky, zatímco u příkazu je to naopak. V některých případech mohou mít i výrazy vedlejší účinky - například přiřazení v C:

```
x = (y = z);
```

výsledkem výrazu y = z je přiřazovaná hodnota a ta je zkopírována také do proměnné x.

Každý programovací jazyk má svá pravidla pro vyhodnocování výrazů. Typicky jsou nejdříve vyhodnoceny operandy (například parametry funkce) a potom je provedena operace (například volání funkce). Pořadí vyhodnocování parametrů nemusí být v některých jazycích definováno - to potom umožňuje implementace pro různé počítače nebo provádění optimalizací. Různé pořadí vyhodnocování podvýrazů může způsobit rozdílné chování programu v případě, že podvýrazy produkují vedlejší efekty, jako v následujícím příkladě:

```
var x : integer;

function f:integer;
begin
   x := x + 1;
   f := x;
end;

function p(a,b:integer):integer;
begin
   p:=a+b;
end;

begin
  x:=1;
  writeln(p(f,x));
end.
```

Pokud jsou parametry při volání funkce p vyhodnoceny zleva doprava, program vypíše hodnotu 4, v opačném případě vypíše 3. Volání funkce f má totiž vedlejší efekt - mění hodnotu

PDF created with FinePrint pdfFactory trial version http://www.fineprint.com

globální proměnné x. Pokud je chování programu závislé na pořadí vyhodnocování operandů a v daném programovacím jazyku není toto pořadí určeno, je takový program nesprávný.

Někdy není nutné vyhodnocovat všechny podvýrazy - například u boolovských binárních operátorů lze použít **zkrácené vyhodnocování výrazů**. Pokud je například první z operandů logického operátoru *or* pravdivý, není nutné vyhodnocovat druhý operand.

3. Formální specifikace sémantiky

Pro některé účely je nutné popsat sémantiku programovacích jazyků formálně. Například pro dokazování programů je to nezbytné. Existuje několik metod formální specifikace sémantiky. V této kapitole budou postupně popsány operační, axiomatická a denotační sémantika.

3.1. Operační sémantika

Operační sémantika definuje sémantiku programovacího jazyka tak, že určí jak je libovolný program vykonán na počítači, jehož činnost je známa. Může to být nějaký abstraktní stroj, který je dostatečně jednoduchý pro snadné pochopení jeho činnosti (například Turingův stroj). Operační sémantika potom specifikuje, jak tento stroj zpracovává program v definovaném jazyku.

Příkladem může být redukční počítač - program je postupně redukován až po získání výsledné hodnoty. Jsou dána redukční pravidla, podle nichž výpočet probíhá. Pro zápis se používají logická inferenční (neboli odvozovací) pravidla - nejdříve je zapsán předpoklad (premisa) a potom pod čarou výsledek. Příkladem může být pravidlo definující tranzitivitu implikace:

$$\frac{a \rightarrow b,\ b \rightarrow c}{a \rightarrow c}$$

Axiom je odvozovací pravidlo bez premisy.

Následující příklad ukazuje gramatiku pro generování jednoduchých výrazů a redukční pravidla pro specifikaci sémantiky. E, E_1 a E_2 jsou výrazy, které ještě nebyly redukovány na hodnoty, V, V_1 a další jsou hodnoty a E $\Rightarrow$ E_1 znamená redukci výrazu E na výraz E_1.

Gramatika: $E \rightarrow E_1$ '+' E_2
| E_1 '-' E_2
| E_1 '*' E_2
| '(' E_1 ')'
| V

Pravidla:
1) V_1 '+' V_2 $\Rightarrow V_1 + V_2$
2) V_1 '-' V_2 $\Rightarrow V_1 - V_2$
3) V_1 '*' V_2 $\Rightarrow V_1 * V_2$
4) '(' V ')' $\Rightarrow$ V

5) $$\frac{E \Rightarrow E_1}{E\ \text{'+'}\ E_2 \Rightarrow E_1\ \text{'+'}\ E_2}$$

6) $$\frac{E \Rightarrow E_1}{E\ \text{'-'}\ E_2 \Rightarrow E_1\ \text{'-'}\ E_2}$$

PDF created with FinePrint pdfFactory trial version http://www.fineprint.com

7) $$\frac{E \Rightarrow E_1}{E \text{ `*' } E_2 \Rightarrow E_1 \text{ `*' } E_2}$$

8) $$\frac{E \Rightarrow E_1}{v \text{ `+' } E \Rightarrow v \text{ `+' } E_1}$$

9) $$\frac{E \Rightarrow E_1}{v \text{ `-' } E \Rightarrow v \text{ `-' } E_1}$$

10) $$\frac{E \Rightarrow E_1}{v \text{ `*' } E \Rightarrow v \text{ `*' } E_1}$$

11) $$\frac{E \Rightarrow E_1}{\text{`(' } E \text{ `)'} \Rightarrow \text{`(' } E_1 \text{ `)'}}$$

12) $$\frac{E \Rightarrow E_1,\ E_1 \Rightarrow E_2}{E \Rightarrow E_2}$$

Pravidla 1 až 4 jsou axiomy. Pokud je mezi dvěmi hodnotami znaménko aritmetické operace, lze tento výraz redukovat na hodnou výsledku dané operace (podle pravidel 1, 2 a 3) a pokud je hodnota v závorce, lze závorky odstranit (podle pravidla 4).

Ostatní pravidla umožňují kombinování jednotlivých redukcí a postupné provádění výpočtu. Pokud je výraz součtem dvou podvýrazů, lze vyhodnotit levý z nich a nahradit jej jeho výsledkem podle pravidla 5. Totéž platí pro rozdíl i součin. Podobně lze vyhodnotit pravý podvýraz, pokud je již levý vypočten (pravidla 8, 9 a 10). Pravidlo 11 říká, že se může vyhodnotit podvýraz uzavřený v závorkách a nakonec pravidlo 12 umožňuje postupné vyhodnocování výrazů.

Dále je potřeba rozšířit operační sémantiku tak aby obsahovala přiřazení a prostředí, například podle následující notace:

$$
\begin{aligned}
P &\rightarrow L \\
L &\rightarrow L_1 \text{ `;' } L_2 \\
  &\quad \mid S \\
S &\rightarrow \text{id} := E
\end{aligned}
$$

Prostředí chápeme jako funkci z množiny identifikátorů do množiny hodnot (v tomto případě celá čísla společně s nedefinovanou hodnotou) .

$$\text{Env: Identifikátor} \rightarrow \text{Integer} \cup \{\text{undef}\}$$

Do všech inferenčních pravidel nyní musíme přidat toto prostředí. Zápis $\langle E|Env\rangle$ potom znamená, že výraz E je vyhodnocen v prostředí Env (to zahrnuje především hodnoty proměnných). Například pravidlo 5 se potom změní takto:

5) $$\frac{\langle E \mid Env\rangle \Rightarrow \langle E_1 \mid Env\rangle}{\langle E \text{ `+' } E_2 \mid Env\rangle \Rightarrow \langle E_1 \text{ `+' } E_2 \mid Env\rangle}$$

Pro vyhodnocení identifikátoru potom musíme přidat následující pravidlo:

PDF created with FinePrint pdfFactory trial version http://www.fineprint.com

$$13)\quad \frac{\mathtt{Env(id)} = \mathtt{v}}{\langle \mathtt{id} | \mathtt{Env} \rangle \Rightarrow \langle \mathtt{v} | \mathtt{Env} \rangle}$$

Zbývá přidat pravidla pro přiřazení a pro zpracování posloupnosti příkazů:

$$14)\quad \langle \mathtt{id:=v} | \mathtt{Env} \rangle \Rightarrow \mathtt{Env}\ \&\ \{\ \mathtt{id} = \mathtt{v}\ \}$$

$$15)\quad \frac{\langle \mathtt{E} | \mathtt{Env} \rangle \Rightarrow \langle \mathtt{E}_1 | \mathtt{Env} \rangle}{\langle \mathtt{id:=E} | \mathtt{Env} \rangle \Rightarrow \langle \mathtt{id:=E}_1 | \mathtt{Env} \rangle}$$

$$16)\quad \frac{\langle \mathtt{S} | \mathtt{Env} \rangle \Rightarrow \mathtt{Env}_1}{\langle \mathtt{S;S}_1 | \mathtt{Env} \rangle \Rightarrow \langle \mathtt{S}_1 | \mathtt{Env}_1 \rangle}$$

Pravidlo 14 určuje zpracování příkazů - výsledkem příkazu přiřazení není hodnota, ale nové prostředí, kde je nahrazena hodnota identifikátoru id.

3.2. Denotační sémantika

Denotační sémantika používá k popisu sémantiky programovacího jazyka funkce. Tyto funkce přiřazují sémantické hodnoty správným syntaktickým zápisům. Příkladem může být funkce Val, která přiřadí výrazu jeho hodnotu:

Val: Výraz → Integer

Doménou syntaktických funkcí se nazývá syntaktická doména - u funkce Val je to množina všech správných aritmetických výrazů. Obor hodnot je sémantická doména - zde množina celých čísel. Denotační definice jazyka má tedy 3 části - syntaktickou doménu, sémantickou doménu a definici jednotlivých funkcí.

Sémantické domény jsou množiny, které ale navíc mohou tvořit strukturu - například celá čísla mají operace +, - a * a dohromady tvoří algebru.

Jako příklad poslouží jednoduchý jazyk definovaný v předchozí kapitole. Pro aritmetické výrazy potom můžeme vytvořit následující denotační sémantiku:

```
Syntaktické domény:      E: Výraz
                         V: Číslo

Sémantické domény:
        V: Integer = { …,-2,-1,0,1,2,… }
  Operace
        + : Integer × Integer → Integer
        - : Integer × Integer → Integer
        * : Integer × Integer → Integer

Sémantické funkce:
```

E: Výraz → Integer

$$\mathbf{E}[[\mathtt{E}_1\ \text{`+'}\ \mathtt{E}_2]] = \mathbf{E}[[\mathtt{E}_1]] + \mathbf{E}[[\mathtt{E}_2]]$$
$$\mathbf{E}[[\mathtt{E}_1\ \text{`-'}\ \mathtt{E}_2]] = \mathbf{E}[[\mathtt{E}_1]] - \mathbf{E}[[\mathtt{E}_2]]$$
$$\mathbf{E}[[\mathtt{E}_1\ \text{`*'}\ \mathtt{E}_2]] = \mathbf{E}[[\mathtt{E}_1]] * \mathbf{E}[[\mathtt{E}_2]]$$
$$\mathbf{E}[[\text{`('}\ \mathtt{E}\ \text{`)'}]] = \mathbf{E}[[\mathtt{E}]]$$

PDF created with FinePrint pdfFactory trial version http://www.fineprint.com

```
E[[V]] = V
```

Podobně jako v předchozí kapitole, nyní přidáme identifikátory, prostředí a příkaz přiřazení. Prostředí jsou funkce z množiny identifikátoru do množiny celých čísel (a navíc nedefinovaná hodnota) a množina prostředí tvoří novou sémantickou doménu:

```
Env: Prostředí = Identifikátor → Integer ∪ {undef}
```

Nedefinovaná hodnota se v denotační sémantice značí symbolem ⊥. Množinu obsahující nedefinovanou hodnotu budeme značit s indexem ⊥, například $Integer_{\perp}$. Při vyhodnocování výrazů nyní musíme předat prostředí jako parametr. Sémantická hodnota výrazu bude proto funkce:

```
E: Výraz → Prostředí → Integer⊥
```

a hodnota identifikátoru je potom:

```
E[[id]](Env) = Env(id)
```

Následuje definice sémantiky pro celý program podle gramatiky z předchozí kapitoly:

```
P: Program → Prostředí
     P[[P]] = L[[L]](Env0)

L: Seznam příkazů → Prostředí → Prostředí
     L[[L1 ';' L2]] = L[[L2]] o L[[L1]]
     L[[S]] = S[[S]]

S: Příkaz → Prostředí → Prostředí
     S[[id := E]](Env) = Env & { I = E[[E]](Env) }
```

3.3. Axiomatická sémantika

Axiomatická sémantika definuje význam programu (nebo jeho části) popisem účinku, který má jeho provedení na různé podmínky. Jedná se o podmínky platné před provedením příkazu (vstupní podmínky) a po něm (výstupní podmínky). Příkladem může být zvýšení proměnné o 1:

```
{x = A}
x := x + 1;
{x = A + 1}
```

Axiomatická specifikace sémantiky programové konstrukce je tedy tvaru

```
{P} S {Q}
```

kde P a Q jsou podmínky a platí, že při platnosti podmínky P před provedením příkazu S je po jeho provedení platná podmínka Q.

Podmínka Q je známa (je to cíl - výsledek, který má mít daný program) a otázka je, jaká je vstupní podmínka. Obecně může být takových podmínek mnoho. Nejdůležitější je takzvaná nejslabší vstupní podmínka. Podmínka P je slabší než jiná podmínka R pokud R → P. Nejslabší podmínku budeme označovat wp (weakest precondition).

PDF created with FinePrint pdfFactory trial version http://www.fineprint.com

Axiomatickou sémantiku jazykové konstrukce S definujeme jako funkci na množině podmínek f(Q) = wp(S,Q). Tato funkce vypočítá nejslabší podmínku, která musí být platná, aby po provedení příkazu S platila podmínka Q. Zde je příklad:

$$wp(x:=1/y,\ x=1/y) = \{y \neq 0\}$$

Nejslabší podmínka má tyto vlastnosti:

$$wp(S,false) = \{false\}$$
$$wp(S,P \wedge Q) = wp(S,P) \wedge wp(S,Q)$$
$$wp(S,P \vee Q) = wp(S,P) \vee wp(S,Q)$$
$$\text{pokud } Q \rightarrow R, \text{ potom } wp(S,Q) \rightarrow wp(S,R)$$

Pro seznam příkazů potom můžeme psát:

$$wp(S_1;\ S_2;\ Q) = wp(S_1,\ wp(S_2,\ Q))$$

Nejslabší vstupní podmínkou seznamu příkazů je tedy kompozice nejslabších podmínek jednotlivých příkazů, přičemž se postupuje od konce seznamu na začátek.

Pro přiřazení platí:

$$wp(id:=E,\ Q) = Q[E/id]$$

kde zápis Q[E/id] znamená, že v podmínce Q se všechny volné výskyty identifikátoru id nahradí výrazem E.

4. Literatura

[1] Louden, K., C., *Programming Languages, Principles and Practice*, PWS Publishing Company, Boston, 1993

PDF created with FinePrint pdfFactory trial version http://www.fineprint.com